Techniques are described herein for using configurable logic constructs in a loop buffer. In an embodiment, a configurable hardware block is programmed based on one or more target functions within a loop. The configurable hardware block is associated with a plurality of registers, including a loopcounter register and a first output register. For each iteration of the loop, a counter value in the loopcounter register is updated and a target value in the first output register is updated using the programmed configurable hardware block. For each iteration of the loop, a set of one or more instructions may be fetched from the instruction buffer and executed based on the updated target value in the first output value.
FIG. 2

INPUT/LOOPCOUNT REGISTER 0 ($r0)

INPUT REGISTER 1 ($r1)

and $r5, $r4, $r3
str $r5, output_array[$r2]

INSTRUCTION FETCH

OUTPUT REGISTER 0 ($r2)

OUTPUT REGISTER 1 ($r3)

204

210

206

208
CONFIGURABLE LOGIC CONSTRUCTS IN A LOOP BUFFER

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is related to application Ser. No. 13/789,524, filed Mar. 7, 2013 the entire contents of which is hereby incorporated by reference as if fully set forth herein.

FIELD OF THE INVENTION

[0002] The present disclosure relates generally to techniques for optimizing loop execution and, more specifically, to techniques for using configurable logic constructs during loop execution.

BACKGROUND

[0003] Many applications spend a large percentage of time in small kernels of instructions that are executed in tight loops. Loop buffers may optimize loop execution in processors that have limited instruction cache and strict power limitations. Specifically, the code kernels that are executed in tight loops are loaded in a loop buffer that is usually much smaller than the instruction cache. Hence, the energy and overhead in fetching instructions is reduced when the loop buffer is utilized. This makes loop buffers useful in power efficient architectures targeted to execute applications that have identified the loop kernels. However, loop buffers are typically very small, limiting their support of code expansion that occurs with loop unrolling.

[0004] For example, the tables below illustrate different approaches for evaluating a loop. Table 1 includes an example code snippet that converts all bits in input integer to bytes in array output_array.

<table>
<thead>
<tr>
<th>TABLE 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Code snippet</td>
</tr>
</tbody>
</table>
| for (i=31; i>=0; i--){  
| output_array[i] = input_integer & (1 << i);  
| } |

[0005] Table 2 below illustrates an example assembly snippet, herein referred to as the "original assembly snippet." This assembly snippet may be generated by compiling the code snippet in Table 1. No loop unrolling or loop buffer is used, resulting in extra instructions to track the number of iterations and to execute the conditional branch.

<table>
<thead>
<tr>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original assembly snippet</td>
</tr>
</tbody>
</table>
| mov $0, 31  
| ld $r1, &input_integer  
| shl $r2, $r1, $r0  
| str $r2, output_array[$r0]  
| sub $0, $r0, $r9  
| jne @v4, 0, 1  
| } |

[0006] Tables 3 and 4 below illustrate alternative assembly snippets generated from the same code snippet. Table 3 exploits loop unrolling resulting in code-size expansion, and Table 4 exploits a loop buffer. By expanding the code as illustrated in Table 3, the instructions for tracking the number of iterations in the loop and evaluating the conditional branch may be eliminated. However, such code may be too large to be stored in a loop buffer.

<table>
<thead>
<tr>
<th>TABLE 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly snippet that exploits loop unrolling</td>
</tr>
</tbody>
</table>
| mov $0, 32  
| ld $r1, &input_integer  
| shl $r2, $r1, $r5  
| str $r2, output_array[$r5]  
| sub $0, $r0, $r9  
| jne @v4, 0, 1  
| } |

[0007] In Table 4, $s0 is the LOOPCOUNT register and is auto decremented by loop buffer hardware. The loop is completed when $s0 reaches 0. The number of instructions in this example compared to loop unrolling is significantly reduced, which facilitates storage in a loop buffer when space is limited.

<table>
<thead>
<tr>
<th>TABLE 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly snippet that exploits loop buffer</td>
</tr>
</tbody>
</table>
| mov $0, 160  
| ld $r1, &input_integer  
| shl $r2, $r1, $r5  
| str $r2, output_array[$r5]  
| sub $0, $r0, $r9  
| jne @v4, 0, 1  
| } |

[0008] Table 5 illustrates the number of instructions executed in each representation of the code. The number of instructions that are executed may approximate the speed at which the code is executed. Accordingly, loop unrolling assembly was approximately 150% faster than the original assembly snippet while the assembly utilizing the loop buffer was approximately 25% faster than the original.

<table>
<thead>
<tr>
<th>TABLE 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison of assembly snippets</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Table 2:</td>
</tr>
<tr>
<td>Table 3:</td>
</tr>
</tbody>
</table>
TABLE 5—continued

<table>
<thead>
<tr>
<th># of instructions executed in 1 iteration of loop</th>
<th># of instructions executed in 32 iteration of loop</th>
<th>Speedup as compared to original assembly snippet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly snippet that exploits loop buffer</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1.25</td>
</tr>
</tbody>
</table>

[0009] As apparent from Table 5, loop unrolling may achieve greater speedup times than using a loop buffer. However, the loop unrolling assembly snippet does not take advantage of the reduced energy and overhead of fetching instructions provided by the assembly snippet that exploits the loop buffer. Although the loop unrolling may be used in conjunction with the loop buffer, the code expansion inherent in loop unrolling may overflow the loop buffer, reducing or negating the benefit of a loop buffer.

[0010] The approaches described in this section are approaches that could be pursued, but not necessarily approaches that have been previously conceived or pursued. Therefore, unless otherwise indicated, it should not be assumed that any of the approaches described in this section qualify as prior art merely by virtue of their inclusion in this section.

SURTIN DESCRIPTION OF THE DRAWINGS

[0011] Various embodiments are illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

[0012] FIG. 1 is a block diagram depicting an example loop buffer with configurable logic constructs, according to an embodiment;

[0013] FIG. 2 is a block diagram depicting an example state of a loop buffer with configurable logic constructs, according to an embodiment;

[0014] FIG. 3 is a block diagram depicting an example state of configurable logic constructs including a tree of operators, according to an embodiment; and

[0015] FIG. 4 is a block diagram depicting a computer system upon which an embodiment may be implemented.

DETAILED DESCRIPTION

[0016] Techniques are described herein for using configurable logic constructs in a loop buffer. In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form in order to avoid unnecessarily obscuring the present invention.

General Overview

[0017] According to embodiments described herein, a configurable hardware block is programmed with loop functions. The configurable hardware block is further associated with a plurality of registers, including a loopcount register, one or more input registers, and one or more output registers. During loop execution, a target value is updated in the output register using the programmed configurable hardware block. In addition, a counter value that tracks a number of iterations in the loop is updated in the loopcount register. The counter value may also be updated by the configurable hardware block per iteration or may be updated by other circuitry.

[0018] During loop execution, a set of instructions may be fetched from an instruction buffer per iteration of the loop. The instructions may be executed based on one or more updated target values stored in the output registers. By relying on the configurable hardware block to update the output registers, the number of instructions stored by the instruction buffer and executed each loop iteration may be significantly reduced, decreasing the storage overhead on the instruction buffer and the execution time of the loop.

System Architecture

[0019] FIG. 1 is a block diagram depicting an example loop buffer with configurable logic constructs, according to an embodiment. System 100 generally comprises one or more input registers 102a to 102n, a loopcount register 104, configurable hardware block 106, configuration data 108, instruction buffer 110, instruction data 112 and output registers 114a to 114n.

[0020] Input registers 102a to 102n store input data that is provided to configurable hardware block 106. In an embodiment, each input register is associated with a different general purpose register (GPR) of a central processing unit (CPU). For example, a first input register may be associated with GPR Sr0 and a second input register may be associated with GPR Sr1. During loop execution, the input data is loaded from the associated GPRs into configurable hardware block 106.

[0021] Loopcount register 104 is an input register that tracks the number of iterations during a loop. This register may also be associated with a GPR of the CPU. Loopcount register 104 stores a counter that is automatically incremented or decremented for each iteration of a loop. Once the value in this register indicates that the loop has finished, the program flow exits the loop body.

[0022] Configurable hardware block 106 includes hardware logic that is reconfigurable by software. The underlying reconfigurable hardware constructs may vary from implementation to implementation. For example, configurable hardware block 106 may comprise a plurality of programmable logic blocks that are coupled together via a programmable interconnect. The logic blocks may comprise programmable look-up tables, logic gates, and/or arithmetic logic units. Configurable hardware block 106 receives configuration information from configuration data 108. Based on configuration data 108, configurable hardware block 106 configures the datapath between input registers 102a to 102n and output register 114a to 114n.

[0023] Configuration data 108 includes configuration information to program configurable hardware block 106. For example, configuration data 108 may include a bitstream that reconfigures a programmable interconnect to change how a plurality of logic blocks are connected. In another example, configuration data 108 may configure the lookup tables for a particular logic blocks to change the output of a given logic block for a given input. Thus, configuration data 108 may control and alter the datapath (i.e., the collection of functional units) between the input registers and the output registers.
Instruction buffer 110 is a buffer memory that stores a set of one or more instructions specified in instruction data 112. During loop execution, the CPU fetches the set of one or more instructions from instruction buffer 110 rather than fetching the instruction from main memory or an instruction cache. Instruction buffer 110 may be a smaller, faster memory cache than a typical instruction cache to minimize latency in fetching loop instructions.

Instruction data 112 comprises a set of one or more instructions that are part of an instruction set architecture (ISA). For example, instruction data 112 may include machine code that may be executed directly by a CPU. Accordingly, each instruction may comprise one or more opcode fields that identify a particular operation supported by the ISA. Each instruction may further comprise one or more other fields used to perform the particular operation.

Output registers 114a to 114a store output data generated by configurable hardware block 106 during loop execution. In an embodiment, each output register is associated with a different GPR. For example, a first output register may be associated with GPR $r2 and a second register may be associated with GPR $r3. During loop execution, the output data is loaded from the configurable hardware block 106 into the associated GPRs.

The associations between registers and the configurable hardware block 106 may be predefined in the hardware architecture or may be software-configurable, depending on the particular implementation. For example, GPRs may be communicatively coupled via a bus to the appropriate registers as dictated by the architecture. In another example, software instructions may be used to configure the connection between the GPRs and configurable hardware block 106.

The number of input and output registers may vary from implementation to implementation. In one embodiment, the number of registers is determined based on the code kernel that is executed by configurable hardware block 106. For example, if the code kernel generates one output from two inputs, then configurable hardware block 106 may use two GPRs for input and one GPR for output. Thus, the number of associated registers may be dynamically determined based on the number and type of loop operations performed by configurable hardware block 106.

Programming Configurable Constructs in a Loop Buffer

In an embodiment, software is used to program the configurable loop buffer based on one or more functions included in a loop. The functions that are programmed into the configurable loop buffer are herein referred to as “target functions.” The target functions may include, without limitation, arithmetic operations, bitwise operations, and logical operations. For example, in the code snippet of Table 1, the loop may be broken down into multiple operations. One operation is the bitwise shift (1<<i), which shifts a bit to the left based on the loop count value. Another operation is the bitwise AND between input_integer and the result of the bitwise shift operation. Yet another operation is decrementing the current loop count to identify a location to store input_integer in the output array. The CPU may program configurable hardware block 106 to evaluate such target functions.

A compiler, such as an assembler, may be used to detect which functions within a loop body should be programmed into configurable hardware block 106 (i.e., which functions are target functions). The manner in which the functions are identified may vary from implementation to implementation. For example, the target functions may be explicitly specified as an assembly directive, such as illustrated in Table 6 below.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov $r0, 32</td>
<td># LOOP_HW_IN0 = $r0</td>
</tr>
<tr>
<td>mov $r1, 1</td>
<td># LOOP_HW_IN1 = $r1</td>
</tr>
<tr>
<td>ld $r4, &amp;input_integer</td>
<td># $r4 = input_integer</td>
</tr>
<tr>
<td>#CONFIG LOOP_HW_OUT0 =</td>
<td></td>
</tr>
<tr>
<td>LOOP_HW_IN0 = LOOP_HW_IN1</td>
<td># $r2 = $r0 + 1</td>
</tr>
<tr>
<td>#CONFIG LOOP_HW_OUT1 =</td>
<td></td>
</tr>
<tr>
<td>LOOP_HW_IN1 = (LOOP_HW_IN0 - LOOP_HW_IN1)</td>
<td></td>
</tr>
<tr>
<td>/$r3 = 1 &lt;&lt; ($r0 + 1)</td>
<td></td>
</tr>
<tr>
<td>loopstart $r5</td>
<td># $r5 shall auto decrement in each iteration</td>
</tr>
<tr>
<td>and $r5, $s4, $r3</td>
<td># $r5 = input_integer &amp; 1 &lt;&lt; i</td>
</tr>
<tr>
<td>str r5, output_array[$r2]</td>
<td># output_array[i] = $r5</td>
</tr>
</tbody>
</table>

The compiler translates the expressions specified by the #CONFIG directive into instructions for programming configurable hardware block 106. For example, the assembly code in Table 6 may be translated as follows: Input Register 0 = LOOP_HW_IN0 = $r0; number of iterations = 32; Input Register 1 = LOOP_HW_IN1 = $r1; Output Register 0 = LOOP_HW_OUT0 = $r2; (32-1); and Output Register 1 = LOOP_HW_OUT1 = $r3 = $r5 $r2. During translation, the compiler generates configuration data 108 that is used to program configurable hardware block 106. In the example illustrated in Table 6, configuration data 108 may be used to program configurable hardware block 106 to decrement the value of $r0 and store the result in $r2. Similarly, configuration data 108 programs configurable hardware block 106 to perform a bitwise left-shift operation on the value stored in $r1 by (32-1) bits. The CPU may use configuration data 108 to program configurable hardware block 106 at assembly time, during program startup, or during program runtime, depending on the particular implementation.

The format of configuration data 108 may vary from implementation to implementation. In an embodiment, configuration data 108 is associated with a memory mapped special register. To reconfigure the logic within configurable hardware block 106, the CPU generates and stores a code in the special register. The code may be generated at compilation time and stored in the memory mapped register at runtime to dynamically program configurable hardware block 106. Each code uniquely identifies how the configurable hardware block 106 is setup and may be generated based on the target functions and the underlying constructs of configurable hardware block 106. For example, special register may be coupled to a plurality of multiplexors that control the interconnects and logical units within configurable hardware block 106. The code may comprise a set of bits that, when stored within the special register, enable or disable the appropriate multiplexors to perform the target logical expressions. A load from the memory mapped register may be performed to identify the current configuration of the hardware. Table 7 illustrates example instructions that may be executed by the CPU to program configurable hardware block 106 at runtime. By exposing expanded views of the loop count variable in the...
output register, the benefits of loop rolling may be achieved. The expanded views can be exposed by configuring the hardware appropriately. This is one sample use case of the configurable hardware.

**TABLE 7**

Assembly snippet for programming a loop buffer

```assembly
mov $r9, 0xABC0  # Move code to GPR $r9
at $r9, 0, mem_addr*  # Store code to memory mapped mem_addr*/
```

[0034] In another embodiment, configuration data 108 is encoded in bitstream formats similar to the format used to configure field programmable gate arrays (FPGAs). The compiler generates the bitstream patterns for the target functions, such as those within the #CONFIG pragmas in the example above. The LOOPSTART instruction points to the address storing the bitstream. During runtime, the bitstream is used to configure the hardware logic at runtime.

[0035] In another embodiment, the CPU may support an extension to its instructions set architecture (ISA). The ISA extension may be invoked from the #CONFIG pragmas in the examples given above. For example, the CPU may execute instructions such as isu_config_special_config_reg_1 0xABC0 to program a special-purpose configuration register with a code that configures the hardware logic of configurable hardware block 106.

[0036] Referring again to Table 6, the loopstart instruction identifies the loopcount register. In the present example, Srl0 is the loopcount register, which also serves as one of the input registers. Accordingly, in addition to tracking the number of iterations, the value in loopcount register 104 may be used to evaluate target functions and update one or more output registers. Alternatively, loopcount register 104 may be separate and independent of input registers 102a to 102n.

[0037] The instructions between the loopstart and loopend instruction identify the instructions that are loaded into instruction buffer 110. The instructions that are loaded into instruction buffer 110 are herein referred to as “target instructions.” For example, Table 8 below illustrates the target instructions for the code snippet of Table 6.

**TABLE 8**

Target instructions for a loop buffer

```assembly
and $r5, $r4, $r3  # $r5 = input_integer & (1 << i)
str $5, output_array[$r2]  # output_array[i] = $r5
```

[0038] In an alternative embodiment, the instruction “and $r5, $r4, $r3” may be programmed into configurable hardware block 106. For example, $r4 may be associated with a third input register labeled LOOP_HW_IN3. Table 9 then illustrates an alternative to the assembly snippet of Table 6. Thus, the number of instructions stored in instruction buffer 110 and executed per loop iteration is reduced, but the number of input registers is increased.

**TABLE 9**

Alternative assembly snippet for programming a loop buffer

```assembly
mov $r0, 32  # LOOP_HW_IN0 = $r0 = LOOPCOUNT = 32
mov $r1, 1  # LOOP_HW_IN1 = $r1 = 1
ld $r4, &input_integer  # $r4 = input_integer
```

[0039] In an embodiment, the compiler reserves the appropriate GPRs in programs that utilize loop buffers. For example, if the program includes a loopstart instruction, the compiler may allocate one or more CPU registers for use with the loop buffer. In the example illustrated in Table 6, the compiler would reserve GPRs $r0 to $r3. However, the number of registers reserved may vary depending on the particular implementation and may be based on the particular code kernel. In programs that utilize the configurable logic, (include the loopstart instruction), the reserved registers are not used while the loop body is being executed using the configurable constructs as these registers are reserved for use by the configurable loop buffer construct. However, programs that do not employ the loop buffer construct may use the reserved GPRs normally.

**Runtime Execution of the Loop Using Configurable Constructs**

[0040] During program runtime, system 100 is used to evaluate the target functions within the loop body. In every iteration of the loop, loopcount register 104 is automatically decremented or otherwise updated. In a similar fashion, configurable hardware block 106 updates output registers 114a to 114d in every iteration of the loop. The instructions stored in instruction buffer 110, if any, may then be fetched and executed based on the updated values in the output registers. An example of this process is illustrated below.

[0041] FIG. 2 is a block diagram depicting a state of system 100 when executing the assembly snippet of Table 6. Before execution of the loop body, the registers are initialized, configurable hardware block 106 is programmed, and instructions data 112 is loaded into instruction buffer 110. For example, to initialize the loopcount register, the CPU executes the instruction “mov $r6, 32”. This loads the number of iterations into loopcount register 104 (i.e., 32). Similarly, the CPU executes the instruction “mov $r1!” to set the initial value of $r1.

[0042] Instructions 210 may be loaded into instruction buffer 110 at any time before loop execution. To load the instruction data, the CPU may execute a LOOP_BUFFER_LOAD instruction to start loading instructions from program memory into the loop buffer. In an alternative embodiment there is no explicit LOOP_BUFFER_LOAD instruction. Rather, after the LOOPSTART instruction is encountered, the processor uses loop buffer as a cache to program memory.

[0043] Configuration data 108 is used to program operators 206 and 208 into configurable hardware block 106. For example, a programmable interconnect may be configured such that input register 202 and 204 are connected to a first logical block corresponding to operator 206. The first logical block is programmed to subtract the value in input register
from input register 202 (i.e., Sr0–Sr1). The interconnect may further be programmed to store the result in output register 212 (Sr2). The same result may also be used to update input register 202, which also serves as the loopcount register. In a similar manner, configuration data 108 may program the interconnect such that the result from the first logical block and the value in input register 204 are connected as input to a second logical block corresponding to operator 208. The second logical block is programmed to evaluate the expression Sr1<<$(S0–S1). The result is stored in register Sr3.

[0044] Once all registers are initialized, the instructions are loaded into the instruction buffer, and the hardware logic is configured, loop execution may begin. The target functions are performed for each iteration of the loop to update the respective output registers. For example, the value of output register 212 (Sr2) would be 31 during the first iteration, 30 during the second iteration, and so forth. Similarly, the value of output register 214 (Sr3) is updated each loop iteration.

[0045] After the output registers are updated for a particular loop iteration, the CPU fetches instructions 210 from instruction buffer 110 and executes these instructions in sequential order. The instructions are executed based on the updated values stored in the output registers. In the present example in Table 6, the instruction “andSr5, Sr4, Sr3” is fetched and executed followed by instruction “str r5, output_array[Sr2]” for each iteration of the loop.

[0046] During each loop iteration, an iteration counter is updated in the LOOPCOUNT register. The logic for updating the LOOPCOUNT register may be programmed into configurable hardware block 106 or may be predefined circuitry separate from configurable hardware block 106, depending on the particular implementation. For example, predefined circuitry may increment or decrement the loop counter by one for each loop iteration. Alternatively, configuration data 108 may program configurable hardware block 106 to track the iteration count.

[0047] By programming configurable hardware block 106, a stride associated with the loop may change from loop to loop. For example, the loop in Table 6 has a unit stride where the loop counter is updated by one unit each iteration of the count. A second loop may have a non-unit stride, such as in the loop illustrated in Table 10 below.

| TABLE 10 |
| Loop with non-unit stride |

```
for (i=0; i<SIZE; i+=8) {
    //Loop Body
}
```

For this loop, configurable hardware block 106 may be programmed using the directive illustrated in Table 11.

| TABLE 11 |
| Directive for programming configurable hardware to track loop iterations with non-unit stride |

```
$sr = (SIZE%8)/((SIZE>>3) + 1); 
$LOOP_HW_IN0 = $sr; // # of iterations
$CONFIG $LOOP_HW_OUTO = 0x18:
$LOOP_HW_IN0 <= 5;
```

[0048] Once the loop count value indicates that the loop has completed all of its iterations (e.g., LOOPCOUNT reaches 0), the program flow exits from the loop body. Hence, the CPU no longer fetches instructions from instruction buffer 110. The CPU may instead fetch the next instruction from main memory or some other program memory as indicated by the CPU’s program counter.

Table 12 below compares the number of instruction executed in assembly when the loop buffer with a configurable hardware block is exploited as illustrated in FIG. 2 with the number of instructions executed for the other snippets of assembly code.

| TABLE 12 |
| Assembly snippet exploiting a loop buffer and a configurable hardware block |

<table>
<thead>
<tr>
<th># of instructions executed in 1 iteration of loop</th>
<th># of instructions executed in 32 iteration of loop</th>
<th>Speedup as compared to original assembly snippet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2: Original snippet</td>
<td>5</td>
<td>160</td>
</tr>
<tr>
<td>Table 3: Snippet that exploits loop unrolling</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>Table 4: Snippet that exploits loop buffer</td>
<td>4</td>
<td>128</td>
</tr>
<tr>
<td>Table 6: Snippet that exploits loop buffer with configurable hardware block</td>
<td>2</td>
<td>64</td>
</tr>
</tbody>
</table>

Generating a Tree of Operators in the Configurable Constructs

[0050] Configurable hardware block 106 may be programmed to form a tree of operators that are executed per loop iteration. Each operator in the tree corresponds to one or more target functions within a loop and may be implemented by a configurable logic block. For example, each configurable logic block may be programmed to perform an arithmetic operation, bitwise operation, or any other combinational logic function of a plurality of inputs to generate a particular result. The result of one operator may then be passed as input to one or more subsequent operators in the chain.

[0051] FIG. 3 is a block diagram depicting an example state of configurable logic constructs including a tree of operators, according to an embodiment. Based on the directive “#CONFIG LOOP_HW_OUTO=(ILOOP_HW_IN0<<5) & LOOP_HW_IN1)=ILOOP_HW_IN3)”, configurable hardware block 106 is programmed with a tree of operators, including operators 302, 304, and 306. Operator 302 receives input from register LOOP_HW_IN0 and LOOP_HW_IN2 and performs a left shift operation. The result of this operation and the value stored in register LOOP_HW_IN1 serve as input for operator 304. Operator 304 performs a bitwise AND operation to generate a second result. The second result and the value stored in LOOP_HW_IN3 serve as input to operator 306. This operator performs a bitwise OR operation to generate a final result, which is stored in register LOOP_HW_OUT0. Thus, the tree of operators 302, 304, and 306 are used to generate a final result; this tree is executed per loop iteration to update the final result stored in the output register.
Dual Purpose Registers

In some instances, a loop parameter that serves as input may be updated in each iteration of the loop, similar to the loop counter. For example, an input value may be multiplied by a scaling constant for each iteration of the loop. In such a case, a particular GPR may be associated with both an input and an output register. Continuing with the present example, an initial input value is loaded from the GPR into configurable hardware block 106 at the beginning of a loop iteration. Configurable hardware block 106 then applies the scaling constant to generate an output value, which is subsequently stored in the same GPR. This updated value is then used as input for the next iteration of the loop. This process continues until the program flow exits the loop.

Hardware Overview

According to one embodiment, the techniques described herein are implemented by one or more special-purpose computing devices. The special-purpose computing devices may be hard-wired to perform the techniques, or may include digital electronic devices such as one or more application-specific integrated circuits (ASICs) or FPGAs that are persistently programmed to perform the techniques, or may include one or more general purpose hardware processors programmed to perform the techniques pursuant to program instructions in firmware, memory, other storage, or a combination thereof. Special-purpose computing devices may also combine custom hard-wired logic, ASICs, or FPGAs with custom programming to accomplish the techniques. Special-purpose computing devices may be desktop computer systems, portable computer systems, handheld devices, networking devices or any other device that incorporates hard-wired and/or program logic to implement the techniques.

For example, FIG. 4 is a block diagram that illustrates a computer system 400 upon which an embodiment of the invention may be implemented. Computer system 400 includes a bus 402 or other communication mechanism for communicating information, and a hardware processor 404 coupled with bus 402 for processing information. Hardware processor 404 may be, for example, a general purpose microprocessor.

Processor 404 may include configurable loop constructs (not shown) for evaluating operations within a loop. The configurable loop constructs include software-configurable hardware, such as configurable hardware block 106, and may be in the pipeline of processor 404. Accordingly, instructions executing on processor 404 may generate configuration data to program target functions into the configurable hardware. The configurable loop constructs may further comprise an instruction buffer that stores one or more instructions within a loop that are native to processor 404. Processor 404 may fetch these instructions during loop execution.

Computer system 400 also includes a main memory 406, such as a random access memory (RAM) or other dynamic storage device, coupled to bus 402 for storing information and instructions to be executed by processor 404. Main memory 406 also may be used for storing temporary variables or other intermediate information during execution of instructions to be executed by processor 404. Such instructions, when stored in non-transitory storage media accessible to processor 404, render computer system 400 into a special-purpose machine that is customized to perform the operations specified in the instructions.

Computer system 400 further includes a read only memory (ROM) 408 or other static storage device coupled to bus 402 for storing static information and instructions for processor 404. A storage device 410, such as a magnetic disk, optical disk, or solid-state drive is provided and coupled to bus 402 for storing information and instructions.

Computer system 400 may be coupled via bus 402 to a display 412, such as a cathode ray tube (CRT), for displaying information to a computer user. An input device 414, including alphanumeric and other keys, is coupled to bus 402 for communicating information and command selections to processor 404. Another type of user input device is cursor control 416, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor 404 and for controlling cursor movement on display 412. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), that allows the device to specify positions in a plane.

Computer system 400 may implement the techniques described herein using customized hard-wired logic, one or more ASICs or FPGAs, firmware and/or program logic which in combination with the computer system causes or programs computer system 400 to be a special-purpose machine. According to one embodiment, the techniques herein are performed by computer system 400 in response to processor 404 executing one or more sequences of one or more instructions contained in main memory 406. Such instructions may be read into main memory 406 from another storage medium, such as storage device 410. Execution of the sequences of instructions contained in main memory 406 causes processor 404 to perform the process steps described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions.

The term “storage media” as used herein refers to any non-transitory media that store data and/or instructions that cause a machine to operate in a specific fashion. Such storage media may comprise non-volatile media and/or volatile media. Non-volatile media includes, for example, optical disks, magnetic disks, or solid-state drives, such as storage device 410. Volatile media includes dynamic memory, such as main memory 406. Common forms of storage media include, for example, a floppy disk, a flexible disk, hard disk, solid-state drive, magnetic tape, or any other magnetic data storage medium, a CD-ROM, any other optical data storage medium, any physical medium with patterns of holes, a RAM, a PROM, and an EPROM, a FLASH-EPROM, NVRAM, any other memory chip or cartridge.

Storage media is distinct from but may be used in conjunction with transmission media. Transmission media participates in transferring information between storage media. For example, transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise bus 402. Transmission media can also take the form of acoustic or light waves, such as those generated during radio-wave and infra-red data communications.

Various forms of media may be involved in carrying one or more sequences of one or more instructions to processor 404 for execution. For example, the instructions may initially be carried on a magnetic disk or solid-state drive of a
remote computer. The remote computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system 400 can receive the data on the telephone line and use an infra-red transmitter to convert the data to an infra-red signal. An infra-red detector can receive the data carried in the infra-red signal and appropriate circuitry can place the data on bus 402. Bus 402 carries the data to main memory 406, from which processor 404 retrieves and executes the instructions. The instructions received by main memory 406 may optionally be stored on storage device 410 either before or after execution by processor 404.

[0063] Computer system 400 also includes a communication interface 418 coupled to bus 402. Communication interface 418 provides a two-way data communication coupling to a network link 420 that is connected to a local network 422. For example, communication interface 418 may be an integrated services digital network (ISDN) card, cable modem, satellite modem, or a modem to provide a data communication connection to a corresponding type of telephone line. As another example, communication interface 418 may be a local area network (LAN) card to provide a data communication connection to a compatible LAN. Wireless links may also be implemented. In any such implementation, communication interface 418 sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

[0064] Network link 420 typically provides data communication through one or more networks to other data devices. For example, network link 420 may provide a connection through local network 422 to a host computer 424 or to data equipment operated by an Internet Service Provider (ISP) 426. ISP 426 in turn provides data communication services through the world wide packet data communication network now commonly referred to as the “Internet” 428. Local network 422 and Internet 428 both use electrical, electromagnetic or optical signals that carry digital data streams. The signals through the various networks and the signals on network link 420 and through communication interface 418, which carry the digital data to and from computer system 400, are example forms of transmission media.

[0065] Computer system 400 can send messages and receive data, including program code, through the network (s), network link 420 and communication interface 418. In the Internet example, a server 430 might transmit a requested code for an application program through Internet 428, ISP 426, local network 422 and communication interface 418.

[0066] The received code may be executed by processor 404 as it is received, and/or stored in storage device 410, or other non-volatile storage for later execution.

Extensions and Alternatives

[0067] In the foregoing specification, embodiments of the invention have been described with reference to numerous specific details that may vary from implementation to implementation. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense. The sole and exclusive indicator of the scope of the invention, and what is intended by the applicants to be the scope of the invention, is the literal and equivalent scope of the set of claims that issue from this application, in the specific form in which such claims issue, including any subsequent correction.

What is claimed is:

1. A method for executing a loop on a computing system, the method comprising:
programming a configurable hardware block based on one or more target functions within the loop;
wherein the configurable hardware block is associated with a plurality of registers;
wherein the plurality of registers includes a loopcounter register, an input register and an output register;
for each iteration of the loop:
(a) updating, in the loopcounter register, a counter value that tracks a number of iterations in the loop; and
(b) updating, in the output register based at least in part on a value in the input register, a target value using the programmed configurable hardware block.

2. The method of claim 1, further comprising:
storing a set of one or more instructions within the loop in an instruction buffer;
for each iteration of the loop:
(a) fetching the set of one or more instructions from the instruction buffer; and
(b) executing the set of one or more instructions based on the updated target value in the output register.

3. The method of claim 2, further comprising:
determining, based on the counter value, that a program flow should exit the loop;
in response to determining, based on the counter value, that the program flow should exit the loop, fetching a subsequent instruction in the program flow from a different location than the instruction buffer.

4. The method of claim 1, wherein the input register is the loopcounter register.

5. The method of claim 1, wherein the input register provides at least one input value to the programmed configurable hardware block for each iteration of the loop.

6. The method of claim 1, wherein each register of the plurality of registers is a general purpose register (GPR) of a central processing unit (CPU) within the computing system; wherein updating, in the output register, a target value using the programmed configurable hardware block comprises updating the respective GPR.

7. The method of claim 1, wherein the plurality of registers further includes a second output register, the method further comprising updating, in the second output register, a second target value using the programmed configurable hardware block.

8. The method of claim 7, wherein the second target value is updated, based at least in part, on the value in the input register.

9. The method of claim 1, wherein programming a configurable hardware block based on one or more target functions within the loop comprises forming a tree of operators that are executed for each loop iteration; wherein the output of a first operator in the chain is used as input to a second operator in the chain.

10. One or more non-transitory computer-readable media storing instructions for executing a loop on a computer system, the instructions, in response to being executed by one or more processors, causing the computing system to perform operations comprising:
programming a configurable hardware block based on one or more target functions within the loop;
wherein the configurable hardware block is associated with a plurality of registers;
wherein the plurality of registers includes a loopcount register, an input register, and an output register;  
for each iteration of the loop,  
(a) updating, in the loopcount register, a counter value that tracks a number of iterations in the loop; and  
(b) updating, in the output register based at least in part on a value in the input register, a target value using the programmed configurable hardware block.

11. The non-transitory computer-readable media of claim 10, wherein the instructions further cause operations comprising:  
storing a set of one or more instructions within the loop in an instruction buffer;  
for each iteration of the loop,  
(a) fetching the set of one or more instructions from the instruction buffer; and  
(b) executing the set of one or more instructions based on the updated target value in the output register.

12. The non-transitory computer-readable media of claim 11, wherein the instructions further cause operations comprising:  
(determining, based on the counter value, that a program flow should exit the loop;  
in response to determining, based on the counter value, that the program flow should exit the loop, fetching a subsequent instruction in the program flow from a different location than the instruction buffer.

13. The non-transitory computer-readable media of claim 10, wherein the input register is the loopcounter register.

14. The non-transitory computer-readable media of claim 10, wherein the instructions cause the at least one input register to provide at least one input value to the programmed configurable hardware block for each iteration of the loop.

15. The non-transitory computer-readable media of claim 10, wherein each register of the plurality of registers is a general purpose register (GPR) of a central processing unit (CPU) within the computing system; wherein instructions for updating, in the output register, a target value using the programmed configurable hardware block comprise instructions for updating the respective GPR.

16. The non-transitory computer-readable media of claim 10, wherein the plurality of registers further includes a second output register, the instructions further causing operations comprising updating, in the second output register, a second target value using the programmed configurable hardware block.

17. The non-transitory computer-readable media of claim 16, wherein the second target value is updated, based at least in part, on the value in the input register.

18. The non-transitory computer-readable media of claim 10, wherein instructions for programming a configurable hardware block based on one or more target functions within the loop comprises instructions for forming a tree of operators that are executed for each loop iteration; wherein the output of a first operator in the chain is used as input to a second operator in the chain.

19. A system for executing a loop, the system comprising:  
one or more processors;  
a configurable hardware block that is programmable by the one or more processors;  
wherein the configurable hardware block is associated with a plurality of registers;

20. The system of claim 19, further comprising:  
an instruction buffer;  
wherein the instructions further cause operations comprising:  
storing a set of one or more instructions within the loop in the instruction buffer;  
for each iteration of the loop,  
(a) fetching the set of one or more instructions from the instruction buffer; and  
(b) executing the set of one or more instructions based on the updated target value in the output register.

21. The system of claim 20, wherein the instructions further cause operations comprising:  
determining, based on the counter value, that a program flow should exit the loop;  
in response to determining, based on the counter value, that the program flow should exit the loop, fetching a subsequent instruction in the program flow from a different location than the instruction buffer.

22. The system of claim 19, wherein the input register is the loopcounter register.

23. The system of claim 19, wherein the instructions cause the at least one input register to provide at least one input value to the programmed configurable hardware block for each iteration of the loop.

24. The system of claim 19, wherein each register of the plurality of registers is a general purpose register (GPR) of a processor of the one or more processors; wherein instructions for updating, in the output register, a target value using the programmed configurable hardware block comprise instructions for updating the respective GPR.

25. The system of claim 19, wherein the plurality of registers further includes a second output register, the instructions further causing operations comprising updating, in the second output register, a second target value using the programmed configurable hardware block.

26. The system of claim 25 wherein the second target value is updated, based at least in part, on the value in the input register.

27. The system of claim 19 wherein instructions for programming a configurable hardware block based on one or more target functions within the loop comprises instructions for forming a tree of operators that are executed for each loop iteration; wherein the output of a first operator in the chain is used as input to a second operator in the chain.